Model Transformation Languages under a Magnifying Glass: A Controlled Experiment with Xtend, ATL, and QVT

Regina Hebig
Chalmers | University of Gothenburg
Sweden

Christoph Seidl
Technische Universität Braunschweig
Germany

Thorsten Berger
Chalmers | University of Gothenburg
Sweden

John Kook Pedersen
IT University of Copenhagen
Denmark

Andrzej Wąsowski
IT University of Copenhagen
Denmark

ABSTRACT

In Model-Driven Software Development, models are automatically processed to support the creation, build, and execution of systems. A large variety of dedicated model-transformation languages exists, promising to efficiently realize the automated processing of models. To investigate the actual benefit of using such specialized languages, we performed a large-scale controlled experiment in which over 78 subjects solve 231 individual tasks using three languages. The experiment sheds light on commonalities and differences between model transformation languages (ATL, QVT-O) and on benefits of using them in common development tasks (comprehension, change, and creation) against a modern general-purpose language (Xtend). Our results show no statistically significant benefit of using a dedicated transformation language over a modern general-purpose language. However, we were able to identify several aspects of transformation programming where domain-specific transformation languages do appear to help, including copying objects, context identification, and conditioning the computation on types.

CCS CONCEPTS

• **Software and its engineering → Context specific languages**; *General programming languages*;

KEYWORDS

Model Transformation Languages, Experiment, Xtend, ATL, QVT

ACM Reference Format:
Regina Hebig, Christoph Seidl, Thorsten Berger, John Kook Pedersen, and Andrzej Wąsowski. 2018. Model Transformation Languages under a Magnifying Glass: A Controlled Experiment with Xtend, ATL, and QVT. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3236024.3236046

ESEC/FSE '18, November 4–9, 2018, Lake Buena Vista, FL, USA

ACM ISBN 978-1-4503-5573-5/18/11…$15.00
https://doi.org/10.1145/3236024.3236046

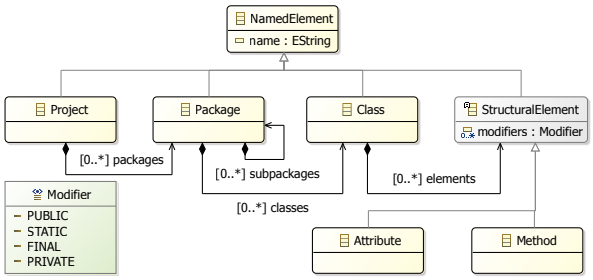

Figure 1: Syntax model for source code

1 INTRODUCTION

In *Model-Driven Software Development* (MDSD) [9, 35, 38] models are automatically processed to support creation, build and execution of systems. Transformations are, among others, used to compute views on models, to validate models [25], to refactor models [43], to interpret or otherwise execute models [28, 30], to implement model debuggers [24], to transform models to lower-level models [31], and to implement model-management operations such as versioning [2].

To support the multitude of processing tasks, many specialized *model-transformation languages* have been developed as a result of a substantial research effort [12, 26]. Outside MDSD, the transformation languages are also gaining importance; in the programming-language [10, 11, 19, 36] and the data-processing community (see for example http://www.cloveretl.com).

Transformation languages come with an implicit promise to be easier to use and more efficient for specifying transformations than general-purpose programming languages (GPLs). However, does this promise hold? We heard this question multiple times in interactions with industrial software teams. Unfortunately, little systematic knowledge exists to respond, besides experience reports from case studies [16]. This hampers choosing between a GPL, which can solve many tasks, and a specialized language that promises efficiency gains, but requires highly specialized programmers.

As the automation of software-engineering tasks is a growing trend, and as software projects mix increasingly many languages and formats, we can only expect that the time spent by developers on creating transformation programs will be growing, and so will the amount and importance of transformations. The transformation languages promise to positively contribute to this effort. It is, thus, key that we develop understanding on how programmers use transformation languages, what problems they face, and what aspects of these languages emerge as particularly useful or problematic.

We settled to answer some of these questions in a controlled experiment with three established *model* transformation languages, employed in program-comprehension, program-modification, and program-creation tasks. We recruited 78 graduate students from Chalmers | University of Gothenburg and Technische Universität Braunschweig, and observed how they cope with the languages: ATL [17], QVT-O [23], and Xtend [8]—the first two dedicated and established model transformation languages, the third a high-level GPL. Analyzing solutions of 231 tasks, we found that:

- Handling multi-valued features (collections), recursion, and designing logical branching conditions are among the most difficult skills to master for the developers.
- Even well qualified subjects struggle to evolve transformations optimally, producing changes of widely diverse sizes.
- Implicit object creation, implicit structure copying, and support for type-driven computation and explicit computation context do appear to reduce the amount of errors.
- There is no sufficient (statistically significant) evidence of a general advantage of specialized model transformation languages (ATL, QVT-O) over a modern GPL (Xtend).

This work aims at the interest of researchers and practitioners working with transformation languages, and building new tools in the modeling, DSL, and programming-languages communities. It can also be of interest to software architects considering using GPLs or dedicated languages for a transformation task.

2 BACKGROUND

We are concerned with *model-to-model* (M2M) transformations, so programs transforming instances of a *source* model to instances of a *target* model (structured data to structured data)—in contrast to model-to-text (M2T) transformations, which are typically implemented in template languages and used for code generation. To represent the abstract syntax of model instances, we use instance specifications (object diagrams), where objects represent model elements, and the links between them represent the relationships between model elements. All three languages considered in this experiment are integrated with the Eclipse Modeling Framework (EMF), which offers a variety of tools and technologies for MDSD based on a simple class diagram language notation called Ecore [39].

Figure 1 shows an example Ecore model[1] of program code, used to implement a refactoring transformation commonly seen in IDEs. Figure 2 shows an example instance specification for this model. The model of Fig. 1 contains classes (called types in the remainder), such as `Project` or `Class`, together with class attributes and class relationships. The latter are either generalizations (e.g., `StructuralElement` is a generalization of `Method`), containment (e.g., `Project.packages`), or association relationships. Containments and associations carry cardinalities, determining how many objects of the type can be in a relationship with the object of the other type (e.g., an object of type `Project` can have 0 or more objects of type `Package`). The instance specification of Fig. 2 contains objects whose structure adheres to the model: links are correct instantiations of the relationships between the objects' types, and each object attribute has a value adhering to the respective class attribute's signature.

[1]In the MDSD community, such an Ecore model is typically called a meta-model.

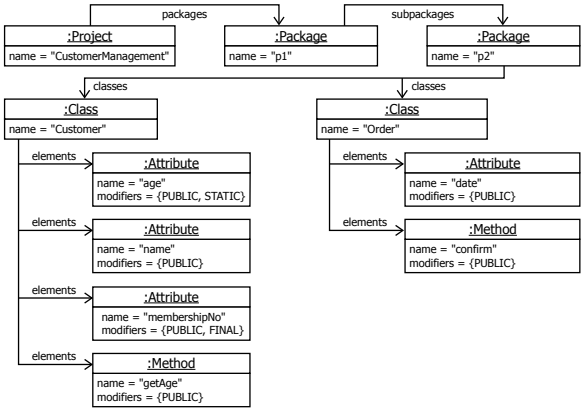

Figure 2: An instance of the model shown in Fig. 1

We focus on three languages used for M2M: The *Atlas Transformation Language (ATL)* is a mostly declarative domain-specific M2M language.[2] Listing 3a shows an example of ATL: The keyword `rule` defines a named transformation for model elements of the source type (`from`) to those of the specified target type (`to`). The first specified rule implicitly designates the entry point for the transformation specification. The invocation order of rules is implicit: The transformation engine processes source instances according to the containment hierarchy specified in the respective model and automatically involves a matching rule for the encountered element of the source instance. The rules marked `lazy` may be invoked manually, in which case the call has to be qualified with the implicit `thisModule` variable. The result of a rule is created by an implicit instantiation of the target type. Then the values are assigned to the fields (object attributes and links) of the instantiated element. In ATL, `<-` serves as the assignment operator. ATL provides various operations on collections: For instance, `exists` checks if at least one collection member satisfies a predicate, `select` creates a derived set with only those members that satisfy a given condition and `collect` (called `map` in many other languages) applies a mapping function to every element of a collection, creating a new collection.

Query-View-Transformation Operational (QVT-O) is an imperative domain-specific M2M language.[3] Listing 3b shows an example in QVT-O: The keyword `main` marks the entry-point for the transformation. The keyword `mapping` defines a named transformation from elements of a source type (specified before the mapping name) to those of a target type (specified after the mapping name). By default, the target type is instantiated automatically when invoking the mapping and the respective element is available via the implicitly initialized variable `result`. Alternatively, it is possible to use the keyword `object` to explicitly instantiate a return type and then assign it to `result` manually. As QVT-O is imperative, the mappings are invoked according to the user-specified control flow. In particular, the `map` operation on collections invokes a specified mapping for each of the source collection's values and stores each transformation result at a similar position in the target collection. In the mapping's body, values are assigned to the fields (object attributes and links) of `result`, which is the implicit context for the

[2]Note that we are aware of ATL's hybrid character due to the imperative nature of its do block in called rules but focus on the commonly used declarative matched rules.
[3]QVT-R is declarative, but we focus on the imperative portion of QVT here.

left-hand side of each assignment. In QVT-O, := is the assignment operator for fields, and += allows to add to collections. QVT-O also has collection operations with similar syntax and semantics as ATL.

Xtend is a GPL with a syntax largely inspired by Java. Unlike ATL and QVT-O, Xtend has no specific language constructs for writing model transformations—additional boilerplate code is used for loading and saving instances. However, thanks to the rich variety of GPL constructs, the language lends itself well to creating model transformations. Listing 3c shows an example of Xtend: The method `main` of an Xtend program constitutes the entry point of a transformation. Further transformation rules can be specified as (usually `static`) methods. The source type that the transformation operates on is specified as the first parameter so that the method may be called as an extension (as if it was defined in the source type). The target is the return type of the method and has to be instantiated manually by calling the respective factory method generated by EMF Ecore. The with-operator => binds the left operand to the implicit context for the sequence of assignments within the lambda expression given in the right operand. Similar to QVT-O, Xtend defines the `map` operation to call a method on all members of a source collection and store the results in a target collection, which is used for explicitly calling transformation methods.

3 PREPARATORY STUDY

We performed a pre-study [33] in collaboration with a Danish industrial partner—a company that consolidates multiple commercial and government data sources to provide support services for trade and taxation in Denmark. The project investigated suitability of model transformation technology in the context of data aggregation. The task involved reading many real-time sources, and correlating transactions in these sources in order to join related records, clean, and normalize the data. The structure of the input and output data has been described in Ecore models. The source model comprised nine classes, 241 attributes and 15 relationships. The target model consisted of two classes, 25 attributes, and one relationship. Creating the output instance required flattening and correlating pieces of information with a complex structure.

The study was performed by a single programmer, the author of the original C# transformation used in the production system and fourth author of this paper, who is a very experienced developer, but new to the transformation languages. At the time, he held nine years of full-time professional programming experience with C#, learned transformation languages in an open university course about MDSD at IT University of Copenhagen, and obtained operational command of them for the purpose of this pre-study.

Two tasks were considered: Create (to study writing a transformation from scratch) and Change (to study evolution of the transformation). For the Create task, the C# transformation has been re-implemented in four languages: ETL [20], the declarative fragment of ATL, Xtend, and Java. As far as possible, the idiomatic style of each language was followed, not the style of the original C# code (except for Java). In the Change task, the transformations have been modified to accept another version of the input model. To ensure that the evolution step was realistic, we used an actual legacy version of the model, slightly older than the one used for the Create task. This ensured that the evolution step was realistic. Although the step was executed 'backwards' from the chronological perspective, the created difference should likely be the same as if it was executed forwards. The difference between the models affected about half of the classes, including dropping classes. The extent of necessary transformation changes was not obvious on the outset.

We performed a qualitative analysis of the created transformations. This list has contributed to the selection of language use dimensions in the design of the actual experiment (Sec. 4).

Observation 1. *The pre-study programs exhibit aspects such as creating objects, populating attributes and other properties with input data, checking logical conditions, filtering and other collection operations involving predicates, ranking and sorting data elements (using standard functions). The Change task involved both restructuring the algorithm of the transformation and performing local modifications such as: renaming of elements, adjusting navigation expressions, etc.*

We counted the sizes of the transformations (see Table 1) after formatting them using rules inspired by the work of Nguyen et al. [29]. Besides the actual transformation, all four languages required implementing startup code, so that they can be executed automatically. The size is included in the table after the plus sign; in Xtend for the Xtend transformation and in Java for the other languages.

Observation 2. *The pre-study transformations implemented in dedicated rule-based languages (ETL, ATL) were up to 48% smaller than the Java[4] variants. For a single nontrivial transformation, the need for startup code reduces the conciseness benefit noticeably.*

We also measured the size of the required change in each of the Change task implementations, see the bottom of Table 1. (The startup code was not modified, so it does not affect the numbers.) Interestingly, the sizes of edits in all languages were extremely close. The size benefit when creating transformations is primarily due to reduction in boilerplate. This boilerplate is relatively robust with respect to changes, and the size benefit disappears in the Change task. This is interesting given that code evolution is commonly accepted as similarly costly, or even more expensive than code creation. For example, Sommerville [37] reports that 65% of effort in software engineering is spend on maintenance and evolution.

Observation 3. *We observed that the size of changes in the pre-study transformations in all involved languages were similar, and the changes almost did not affect boilerplate code.*

Obviously, observations from a single-person single-case experiment are not generalizable due to idiosyncrasy of the case, of the

[4]Java 8 was released only after we had conducted the pre-experiment. Its stream API might reduce the SLOC count of the Java transformation code.

Table 1: Sizes of the pre-study transformations and changes

	ETL	ATL	Xtend	Java
Create				
trafo+startup [loc]	176+82=258	197+79=276	214+71=285	339+73=412
gain vs Java	48%	42%	37%	0%
incl. startup	37%	33%	31%	0%
Change				
diff size [loc]	99	111	103	102
gain vs Java	3%	−9%	0%	0%

```
module RefactoringTrafoATL;
create OUT : MM from IN : MM;


rule Project2Project {
  from s : MM!Project
  to t : MM!Project(
    name <- s.name,
    packages <- s.packages->collect(p|
      thisModule.Package2Package(p)))
}

--...

lazy rule Method2Method {
  from sm : MM!Method
  to tm : MM!Method(
    name<-sm.name,
    modifiers<-sm.modifiers)
}

lazy rule Attribute2Attribute {
   from sa : MM!Attribute
   to ta : MM!Attribute(
    name<-sa.name,
    modifiers<-sa.modifiers)
}
```

(a) ATL

```
modeltype CM uses "ClassModelMM";
transformation RefactoringTrafo
  (in m1:CM, out m2:CM);

main() {
  m1.rootObjects()[Project] ->
    map copyProject();
}

mapping CM::Project::
  copyProject():CM::Project {
  name := self.name;
  packages := self.packages ->
    map copyPackage();
}
//...
mapping CM::StructuralElement::
  copyElement():CM::StructuralElement {
  init {
    if (self.oclIsTypeOf(CM::Method)) {
      result := object CM::Method{};
    } else {
      result := object CM::Attribute{};
    }
  }
  name := self.name;
  modifiers := self.modifiers;
}
```

(b) QVT-O

```
class RefactoringTrafo1 {
val private static factory =
 ClassModelMMFactory.eINSTANCE


def static void main (String[ ] args) {
 // <code to load m1 and save m2 omitted>
 val m2 = copyProject(m1)
}

def static Project copyProject(Project p) {
 factory.createProject => [
  name = p.name
  packages + = p.packages.map[ copyPackage]
 ]}
//...
def static StructuralElement copyElement(Method m) {
 factory.createMethod => [
  name = m.name;
  modifiers + = m.modifiers
 ]}

def static StructuralElement copyElement(Attribute a) {
 factory.createAttribute => [
  name = a.name;
  modifiers + = a.modifiers
 ]}
}
```

(c) Xtend

Figure 3: Excerpts of transformation specifications realizing a refactoring transformation on the source-code model in Fig. 1

programmer, and due to learning effects. Yet, they were sufficiently interesting to motivate further exploration. We proceeded to design a controlled experiment to investigate the matter in detail. We used the pre-study to select the dimensions of interest and to generate some hypotheses. We describe the resulting experiment below.

4 METHOD

We address the following research question: *How effectively programmers use model transformation languages, what problems they face, and what aspects of these languages emerge as particularly useful or problematic in contrast to GPLs?*

4.1 Experiment Design

The treatment in our experiment is the transformation language. Due to their extensive use in education, research, and some industrial applications, we selected ATL as a declarative domain-specific transformation language, QVT-O as an imperative domain-specific transformation language, and Xtend as an imperative GPL. They are all supported by official Eclipse Foundation projects. Finally, we could secure a substantial number of trained subjects for them.

We have decided to not include any pure GPLs. While it would be highly desirable to evaluate functional programming languages in the transformation context (say Haskell), it would also make the experiment considerably more complex, as it is hard to guarantee that students would reach a comparable level of training. Haskell is not well integrated with major modeling tool chains, which would require significant modification of the considered development scenarios, introducing new uncontrolled factors. It would also require substantial extension of the participant base, which is difficult to attain, especially that it is hard to identify a homogeneous population of subjects that know both Haskell and the model transformation languages (functional languages are typically taught in CS degrees, transformation languages in Software Engineering degrees).

The dependent variables are *solution completeness* and *mistakes done* by the participants. We further split these into detailed *dimensions of analysis* arranged by transformation programming skills required for the tasks: *create object*, *initialize single-valued fields*, *initialize collection-valued fields*, *use/interpret value conditions*, *use/interpret type conditions*, *navigate the model*, *place computations in correct context* (Table 3, cf. Obs. 1).

The experiment follows a *between-subjects* design, where each participant learns and works with one of the three languages. We consider three development tasks: transformation *comprehension*, transformation *creation*, and transformation *changing*. These are selected as key activities in code development and evolution. As comprehension is a prerequisite for change, we introduce an explicit dependency (ordering) between the two. The subjects first solve a comprehension task, then they approach a change task for the same code. To avoid learning bias they solve the creation task for another problem. We use two domain scenarios to allow this cross-over. Subjects starting with the first scenario (feature modeling) cross to the second scenario (refactoring) and vice-versa, as Fig. 4 shows:

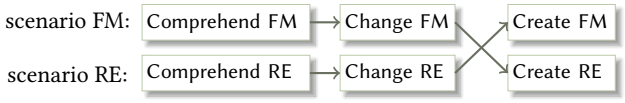

Figure 4: Task execution paths in the cross-over design

Scenarios. The *feature model* (FM) scenario is a transformation between two different representation formats of feature models [18]. FMs are tree-like structures that capture features of a software product line (a portfolio of software variants). As many different variations of the FM notation exist, and even the same notation can be represented differently, creating a transformation to convert between two representations is a realistic scenario [6, 7]. The transformation reorganizes instance objects among the containment

hierarchy differently. Note that we taught feature models in the course to give our subjects sufficient exposition.

The *refactoring* (RE) scenario is the implementation of an attribute refactoring for simple object-oriented syntax model (Fig. 1), a task common in IDEs with support for properties. The source and target models are the same. The task is to create getters and setters for each non-public attribute, depending on the absence of `static` and `final` modifiers. A fragment of this transformation is shown in Fig. 3 for all considered languages.

Tasks. In the *comprehension* task, the subject attempts to understand what a transformation does, e.g., whether parts of a model are copied, modified or new parts created. The subjects get the source and target models and an implementation of a transformation for one scenario in one language (we prepared the implementation of both scenarios in all three languages, in total six programs). We then provide a source instance diagram, and ask to draw the output instance. We assess the correctness of the produced output with respect to the dimensions of analysis listed above (also see Table 3). For instance, if the wrong model elements are modified, we assume that the subject does not command the skill of *use/interpret value conditions*, or the skill of *use/interpret type conditions* (depending on whether the type or logical condition was missed).

The *change* task investigates whether a subject can identify where in the code a change is needed, what information needs to be accessed, and whether she can actually implement the change. We provide the same transformation as for the comprehension task. For the FM scenario, we ask the participants to adapt the transformation to use cardinalities instead of the class hierarchy to represent feature groups (a new target model is provided). For the RE scenario, we ask to amend the transformation to set the visibility modifier of an attribute to private if a getter or setter exists (still the same target model). We assess the correctness of the change with respect to the same dimensions. For instance, if the changed expressions do not follow the types of the target model, we assess that the skill *navigating the model* has not been demonstrated.

Finally, in the *create* task each subject attempts to construct a new transformation from scratch in her language. We provide the relevant source and target models, a description of the requirements in English, as well as stubs for the transformation headers and boilerplate code in the appropriate language (this helps to disambiguate the task). We then ask the subject to produce the transformation. Again, we assess the result using the same dimensions of analysis. For instance, if the produced code places the main computation in a context of a type, from which some relevant information is not accessible, we assess the subject does not demonstrate the skill of *placing computations in a correct context*.

The scenarios, models, transformations, stubs, expected solutions, and task definitions are available in our online appendix [14].

Subject Selection. The subjects are graduate students at the European universities Chalmers | University of Gothenburg (GU) and TU Braunschweig, who just completed a course on MDSD during which they had learned the transformation languages used in this experiment. All had several years of prior experience with programming, but had not worked with transformation languages before the course. Each of the transformation languages was introduced in a separate lecture addressing all participants. Then, each participant implemented three transformations in the transformation language (randomly) assigned to them in the beginning (and later used in the experiment). This included solving and formative assessment of transformations, supported by learning material, examples, and supervision. At the point of the experiment they have been acquainted with the languages for at least one month. Table 2 shows the distribution of participants across project partners.

Table 2: Distribution of subjects across runs and languages

	Chalmers | GU (spring 2016)	TU Braunschw. (fall 2016)	Chalmers | GU (spring 2017)	total
ATL	9	10	7	26
QVT-O	12	8	6	26
Xtend	8	9	9	26
total	29	27	22	78

These subjects are a reasonable representation for programmers, who are considering using transformation languages in a professional setting (are evaluating or starting to use the technology). This is an important target group, as the MDSD technology is in the adoption phase. Results might not transfer to experienced developers using these languages for an extended period.

Controlled Variables. All languages (assigned randomly) are represented in similar proportions in all three runs of the experiment (Table 2). Among others, this avoided an impact of country, teachers, and university on the results. Tasks were assigned randomly: 38 subject have followed the top path in Fig. 4, and 40 subjects have followed the bottom one. Each subject has been asked to attempt all three tasks. Only three participants did not attempt the 3rd task, as they left the experiment early. This resulted in 231 answers, 111 for the first path and 120 for the second path.

4.2 Execution and Data Collection

The participation in the experiment was voluntary and had no impact on the students' grades. We incentivized the participation by emphasizing that the experiment conditions are similar to exam conditions, so it can be used as training. Students were aware of the experiment situation. We briefed the subjects after the experiment to discuss their challenges, with the goal of reducing negative self-perception due to failure in the more challenging tasks.

The tasks were solved on paper, which eliminates the factor of familiarity with the programming environment. It also allows us to investigate the very languages, not the quality of the error reporting from (say) a type checker. The models and transformations were distributed as printouts. The models were small enough to each fit one page. For the change tasks we printed the transformations with increased interline space. For the create tasks we printed stubs with white gaps. To compensate for the fact that subjects work on paper, we provided cheat-sheets that summarized main language features and concepts that participants might want to look up.

The experiment was performed in three runs at two technical universities in two European countries (as summarized in Table 2). Each run of the experiment was performed in a lecture room, where participants had enough space to work. We distributed the tasks in such a way that participants sitting next to each other would solve different tasks. The time limit for the experiment was set to 1 hour and 10 minutes. The subjects were allowed to leave early. We

suggested to use the first 20 minutes for the comprehension task, the next 20 minutes for the change task, and the last 30 minutes for the create task. We announced the current time at the 20 and 30 minute marks. Not all subjects succeeded in completing all the tasks in the allotted time, and some have left early.

We eliminated two entries: one of a student following the course a year earlier, and one of a student with a writing handicap.

4.3 Data Analysis

Scoring. After we collected the data, we graded the points against the dimensions of analysis. Not each dimension is applicable to every task/scenario combination, as not all language elements are always used. Table 3 summarizes the dimensions, and their compatibility with tasks. The dimensions correspond to operational skills to facilitate assessment. Yet, assessing the score of a student's solution does differ somewhat from an exam grading process. For example, for the comprehension task severe syntax problems would, in an exam, lead to no points. We were forgiving to syntax problems if the intention was clear.

We prepared a criterion description for each dimension of analysis in the context of each task. For example for the skill *navigate the model*, for the comprehension task, the description said *Correct overall structure, links, and objects with correct attributes [in the output instance].* In total 23 such descriptions were created,ranging from 10 to 100 words (see online appendix). These descriptions were used to perform the scoring possibly uniformly.

We score each dimension on a three step scale: 0 (no demonstration of the skill), 1 (clear demonstration). The step 0.5 was used when some demonstration of the skill was observed, but we could not agree on assigning 0 or 1. The scoring happened in three phases:

(1) *Pair scoring:* To enable alignment of the scoring styles, three authors have scored a hand-in each (3 tasks) in a think-aloud mode, in pair with another author. Each of the three sessions lasted 90 minutes. The observing author acted as a consultant, asking and answering questions.
(2) *Scoring by language:* The three authors performed the first scoring of all the tasks in one language each. This cost about 2.5 days per person. Besides recording the scores, we annotated the records with qualitative justifications of assigned points, observed mistakes and other points of interest.
(3) *Scoring by task:* Another author (not involved in phase 2) revised all the scores in the by-task direction, to ensure consistency of scoring across languages. This resulted in several corrections in scoring. This allowed to control for different reactions of graders to unexpected mistakes.

None of the authors is involved with the development, or otherwise heavy use of the languages that were subject to this experiment. In the grading, we attempted to be objective as much as possible.

Hypothesis testing. Declarative languages have a lower spread than imperative languages. Therefore, most programmers are rather familiar with the imperative way of thinking. Therefore, we formulate the following hypotheses including null hypotheses:

H1: *Subjects comprehending a QVT-O or Xtend transformation perform **better** than those who comprehend a transformation in ATL.*
H1$_0$: *Subjects comprehending a QVT-O or Xtend transformation perform **worse or equal** to those comprehending an ATL transformation.*

H2: *Subjects who change a transformation written in QVT-O or Xtend perform **better** than those who change a transformation in ATL.*
H2$_0$: *Subjects changeing a transformation written in QVT-O or Xtend perform **worse or equal** to those who change an ATL transformation.*

H3: *Subjects who create a model transformation in QVT-O or Xtend perform **better** than those who create a transformation in ATL.*
H3$_0$: *Subjects who create a model transformation in QVT-O or Xtend perform **worse or equal** to those who create a transformation in ATL.*

As the main purpose of domain specific languages it is to make task solving in a domain easier or more efficient. Thus, we formulate the following hypotheses including null hypotheses:

H4: *Subjects who comprehend a transformation in QVT-O perform **better** than those comprehending a transformation in Xtend.*
H4$_0$: *Subjects who comprehend a transformation written in QVT-O perform **worse or equal** to those comprehending it in Xtend.*

H5: *Subjects changing a transformation written in QVT-O perform **better** than the subjects who change an Xtend transformation.*
H5$_0$: *Subjects changing a transformation in QVT-O perform **worse or equal** than the subjects who change an Xtend transformation.*

H6: *Subjects who create a model transformation in QVT-O perform **better** than those who create a transformation in Xtend.*
H6$_0$: *Subjects who create a model transformation in QVT-O perform **worse or equal** to those who create a transformation in Xtend.*

Analysis. For the quantitative analysis, we compared the average scores and standard deviation for the three languages per task and scenario, using the comprehension tasks for hypotheses 1 and 4, the change tasks for hypotheses 2 and 5, and the creation tasks for hypotheses 3 and 6. We used a Shapiro normality test on each data set (i.e. each set of student's scores for one task and language), which showed that we cannot assume normal distribution for some of the data sets. Therefore, we decided to use the non-parametric Wilcoxon signed-rank sum test. We tested each task to compare the scores reach with the three different languages, using the standard confidence level 0.95. Due to the setup, we test every combination of languages 6 times (two scenarios for each task, comprehend, change, and create). Therefore, we apply a Bonferroni correction [1] to the threshold for the confidence levels, by dividing 0.05 by 6, leading to a threshold of 0.0083. To ensure also relevance, we assessed effect size using Vargha and Delaney's A (VDA), following their interpretation[42]: $A \approx 0.56$ = small; $A \approx 0.64$ = medium; and $A \approx 0.71$ = large. All test were executed with R.

Furthermore, we analyzed the statistics for the individual dimensions of analysis and analyzed (comparing solutions and language constructs) the cases were strong interesting effects were visible.

5 EXPERIMENT RESULTS

Table 4 summarizes the average scores awarded to subjects by task and language. In the following, we elaborate on some of these.

5.1 Analysis of Typing Errors

In the comprehension task, many subjects created instances that are invalid (violate model types). This could be an indicator of insufficient competence of the subjects. In order to ensure that we assess success in applying model transformation skills (as opposed to modeling skills), we wanted to eliminate the subjects who do

Table 3: Applicability of dimensions of analysis among tasks

dimension	description	Comprehend FM	Comprehend RE	Change FM	Change RE	Create FM	Create RE
basic constructs							
initialize single-valued fields	correctly interprets or creates value assignments to single-valued object attributes or links (ATL: <-, QVT-O: :=, Xtend: =)	●	●	●	N/A	●	●
initialize collection-valued fields	correctly interprets or creates value assignments to single-valued object attributes or links (ATL: <-, QVT-O: += or :=, Xtend: += or =)	●	●	N/A	●	●	●
use/interpret value conditions	correctly interprets or uses value conditions	●	●	N/A	●	●	●
use/interpret type conditions	correctly interprets or uses conditions on types	●	●	●	N/A	●	●
navigate the model	correctly interprets or uses object traversal (follow links/access object attributes)	●	●	N/A	●	●	●
compound constructs							
create object	correctly interprets or uses object creation	●	●	●	N/A	●	●
copy complex elements	correctly interprets or creates rules that copy multiple objects with all their attributes and links	●	●	N/A	●	●	●
transformation decomposition							
place computations in correct context	places computation into the correct context	N/A	N/A	●	●	●	●

● Used as grading criteria N/A Not applicable

not demonstrate understanding of the class diagram language. We used *navigate the model* score of the comprehension task to identify *qualified subjects*, interpreting it as a proxy for the skill of modeling. We assume that only subjects with a positive (1 or 0.5) score for this dimension in their comprehension task solution are qualified. So, for this particular analysis in this subsection, we projected the other subjects out of the data set to consider typing errors specific to transformations. This left us with data for 60 subjects (75% of the original sample). As expected, the subjects are still well distributed across all the buckets, at least 8, and at most 12 subjects taking each path in Fig. 4 for each language.

We begin by summarizing the typical type-related mistakes seen:

- Objects are created on the wrong abstraction level, e.g., we found instances of instances.
- Source and target models are merged, e.g., the output instance contained instances of classes from both models.
- Elements from the transformation, e.g., variable names from the transformation program, occur as data in the output
- Associations in the instance do not conform to the model
- Objects are contained by multiple containers, violating the no-sharing constraint of class diagrams
- Instances of non-existing types are created.
- Abstract classes are instantiated

These problems could be observed throughout the three languages with similar intensity (11 times for ATL, 7 for QVT-O, 11 for Xtend). As educators, we find this list useful, as a specification of what training material for teaching transformations shall be created.

5.2 Creating and Copying Objects

Creating objects and understanding what objects are created is a basic, but crucial skill. Interestingly, >80% of the qualified subjects using QVT-O demonstrate the *create object* skill (score 1), while only ca. 60% do this for ATL and Xtend. A reason might be that object creation happens implicitly in QVT-O, with special language support. In Xtend, complex factory methods need to be called explicitly. In ATL, not all blocks can create objects (rules can, but not helpers). This lack of orthogonality may cause some errors.

Observation 4. *Qualified subjects programming QVT-O correctly use and reason about object creation more often (80% of cases) than subjects using ATL and Xtend (60% of cases), across all three tasks.*

Most subjects correctly identify that a transformation copies structures of multiple connected objects. Similarly, most subjects are able to create such copying when writing transformations. Yet, the average score of qualified subjects using our two domain-specific languages is higher (> 0.8) than for our GPL Xtend (0.71).

Observation 5. *Subjects copy complex structures, and reason about copying, more effectively with the domain-specific transformation languages ATL and QVT-O than with the GPL Xtend.*

Copying complex structures does play an important role in model transformations. The experiment confirms, that dedicated support for this in the domain-specific languages is paying well. In contrast, Xtend requires using recursion for the task, which is notoriously difficult. We remark that, in situations that were not handled with automatic copying, so when explicit recursion was required, we observed the same difficulties across all three languages. This indicates that further support for hiding recursion might be useful (for example, the Rascal language [19] has first-class rich visitors to build sophisticated traversals without recursion).

We also investigated how collection-valued fields are handled. In models and transformations, the boundary between simple values and collections is blurred. Simple values are often seen as singleton collections. The GPLs distinguish these firmly. Approximately 66% of qualified subjects master the skill of initializing collection-valued fields. This proportion is lower for ATL (54%) and Xtend (58%). It does appear that this performance is low for all three languages, and there is space for improving collection support in them.

5.3 Identifying the Correct Context

In transformations it is key to identify the location in the model (the type), where the computations are being placed. The context determines which information and operations are reachable via navigation, how far, and also which information is accessible implicitly (via the `this` reference). We record an advantage of subjects working with our domain-specific languages QVT-O and ATL (>60% average score) over those with the GPL Xtend (41% average score).

Observation 6. *Context selection is easier for the subjects working with the domain-specific transformation languages ATL and QVT-O than with the GPL Xtend.*

Table 4: Average scores of subjects per task/language normalized to 1 (includes both qualified and not qualified subjects)

	Comprehend FM			Comprehend RE			Change FM			Change RE			Create FM			Create RE		
	ATL	QVT-O	Xtend	ATL	QVT-O	Xtend	ATL	QVT-O	Xtend	ATL	QVT-O	Xtend	ATL	QVT-O	Xtend	ATL	QVT-O	Xtend
Average	0.63	0.82	0.62	0.42	0.55	0.41	0.55	0.96	0.63	0.23	0.33	0.42	0.59	0.62	0.55	0.53	0.70	0.41
Std.Dev.	0.35	0.29	0.41	0.33	0.32	0.36	0.47	0.10	0.42	0.27	0.26	0.27	0.17	0.26	0.26	0.28	0.31	0.37

Given that only 48% of task solutions score maximum (1) in this dimension, better support for context selection might be a focus for future research and innovation (e.g., opportunities of detecting bad context as smell, and providing automatic refactoring support).

We speculate that an advantage of our domain-specific languages QVT-O and ATL is their encouragement of a better decomposition of the transformation into rules, enhancing comprehension. Specifically, they require to explicitly declare the input and output types instead of resorting to (not even immutable) parameters of a method call in Xtend. Programmers in Xtend might approach the decomposition less consciously. Investigating this aspect in a separate study constitutes valuable future work.

5.4 Branching on Values and Types

Most subjects had difficulty with interpretation or writing branching, selection, or filtering conditions. Remarkably, this skill scored the lowest amongst all: the average score of qualified subjects was 0.32 (with all the other dimensions scoring on average above 0.5).

Observation 7. *Creating correct conditions for branching and object selection (as well as understanding them correctly) is by far the most difficult skill among those we analyzed.*

This observation is both interesting and expected. Expected, as creating the right branching structure for any program is the very essence of the computer programming art, but also one of its key difficulties. Interesting, because this is an aspect to which the transformation language designer community traditionally attaches little attention. The expression sublanguages used within transformation languages tend to be imported from elsewhere (OCL, Java, JavaScript), and little transformation-specific innovation happens. Perhaps, this low score is a pointer for the designers to try again.

There is a notable exception to stagnation in expression languages for transformations: type-dependent branching. Transformation languages have specialized facilities for scheduling computations based on types (usually a form of pattern matching). This is true for all three considered languages. For this reason, we have tracked the use of type conditions separately from predicates on values in the analysis. The average score for using type conditions across all languages and tasks is 0.54 for qualified subjects; substantially more than using value conditions (0.32). This is a good indication that the investment in the language design may pay off.

Observation 8. *Subjects are much more likely to create correct branching conditions based on types than based on values in the transformation specifications.*

5.5 Size of Change Edits (Evolution)

Let us get back to Obs. 3 from the pre-study where we found that the change sizes were similar across the languages. To expand on this observation, we investigated differences in the size of edits that the subjects perform for the change tasks. Working on paper, the subjects did not follow common line breaking conventions, so we measured size differences unambiguously in tokens, not lines.

First, we created specimen solutions to the change tasks. We observed that they had similar diff sizes in all three languages, corroborating Obs. 3. The edit required removal/addition/modification of 20 tokens in ATL, 25 tokens in Xtend, and 19 tokens in QVT-O (cf. task descriptions and example solutions in the online appendix). These numbers are likely close to the optimal minimal change.

For the subject solutions, we manually counted the size of diffs in tokens for all *correct* solutions. We only considered this problem for the Change FM task, as the results of Change RE did not contain sufficiently many correct answers. We assumed that a solution is correct if scored at least 90% of the maximum score summed over all dimensions; 18 subjects have met this condition. We removed two outliers: one with 55 (QVT) and one with 97 tokens (Xtend).

Table 5 summarizes the results. It appears that, even though the specimen solutions do seem to confirm Obs. 3, the average performance of our subjects does not. Also, even if the specimen solution for QVT-O was the smallest, the average subject solution for QVT-O is the largest, and the QVT-O sample is characterized by the largest standard deviation. This might be yet another confirmation of the fact that even if the language expert can present a perfect use of the language constructs, the users may struggle to exploit the opportunities in the language to the same extent. Only for ATL the optimal solution is well approximated (but this has to be interpreted with care, as we are down to only 4 data points in this case).

Observation 9. *Sizes of the change edits performed by subjects differ widely across the population and are most often far from the optimal solutions, not exploiting fully the potential of the languages.*

A possible explanation for the phenomenon is that the specimen and the pre-study solutions were implemented by more experienced programmers than our subjects are. This topic requires a deeper investigation. It would be interesting to study productivity differences between programmers using GPLs and specialized transformation languages, and between senior and junior developers in the context of transformation. Otherwise, it is difficult to convince larger parts of the industry to adopt these technologies.

5.6 Hypothesis Testing

We now return to our hypotheses, investigating whether the overall differences in scoring between languages are statistically significant. Table 6 summarizes the test results. We split the tests by task and scenario to reflect the differences in the scenarios.

Table 5: Size of the change edit for Create FM task

	ATL	QVT-O	Xtend
average diff size [tokens]	18.5 ± 1.0	28.1 ± 8.6	26.3 ± 5.7

After the Bonferroni correction no statistically significant difference can be found after the Bonferroni correction. Most differences seem to be of small to negligible effect size.

Consequently, we cannot reject the null-hypotheses that ATL is leading to a similar or better performance than QVTO and Xtend (H1, H2, and H3 for each task respectively). More interestingly, we cannot reject the null-hypothesis that the domain-specific language QVTO perform equal or worse than the general purpose language Xtend (H4, H5, and H6 for each task respectively). Which leads us to the following controversial observation:

Observation 10. *There is no sufficient (statistically significant) evidence of general advantage of the specialized model transformation language QVTO over the modern GPL Xtend for the tasks, scenarios, and analysis dimensions considered in this paper.*

This is clearly concerning, given the amount of work that has been invested in model transformation technology. We do emphasize that this conclusion is made under narrow conditions, and encourage others to corroborate or reject our observation via further studies.

6 THREATS TO VALIDITY

Construct Validity. We conceived the tasks carefully and ensured that they are solvable with similar effort in all languages. The ATL tasks were limited to the declarative part of ATL; imperative concepts were neither used nor taught. We scored measurable skills instead of the pure completeness of solutions, clearly defining scoring criteria measuring the most general aspects of the languages. While we could conceive the comprehension and creation tasks to cover all general aspects, this was difficult for the change tasks. So, we designed the latter differently for the two scenarios, so that they together cover all aspects.

We admit the limitations of a pen-and-paper experiment. Whether to study pure concepts or concepts embedded in tooling is a standard problem in designing experiments. The former allows to control the conditions better, the latter provides a more realistic setting, but also many confounding factors: familiarity with tools and IDE, stability, ad hoc influences such as screen size, etc. Ideally, both kinds of experiments are needed to obtain a complete picture. In this paper, we decided to study the ability to comprehend and express transformations as a function of the concepts of transformation languages. This has the advantage of producing more fundamental durable findings (languages and language groups change much slower than tools). However, this means that our conclusions should not be generalized to tools. We did mitigate this limitation somewhat by ignoring simple errors, such as typos and minor syntactic problems, which would be detected by tools.

Internal Validity. The perception of a teacher being (dis)passionate about a language may influence subject's performance. To mitigate this we used two locations with different teachers. Differences caused by the university, local culture, and the teacher were controlled by enforcing an equal distribution of the treatments in both sites. The imbalance in the number of data-points between the universities could be problematic if the results were differed. Fortunately, no big difference were observed.

We mitigated selection bias by randomizing the student-treatment assignment early in the courses. Yet, some students left before the experiment (late in the course), but the mortality rate was low and affected all languages similarly. We did not count these cases as failed attempts. We prohibited communication and assigned tasks with different scenarios to adjacent participants.

Table 6: Wilcoxon rank sum test (significance level: 0.5%, corrected threshold for p-values: 0.0083)

x	y	h_0	p-value	result	VD.A
Comprehend FM					
QVTO	ATL	x ≤ y	0.036	h_0 not rejected	0.70 medium
QVTO	Xtend	x ≤ y	0.074	h_0 not rejected	0.66 small
Xtend	ATL	x ≤ y	0.489	h_0 not rejected	0.49 neg.
Comprehend RE					
QVTO	ATL	x ≤ y	0.15	h_0 not rejected	0.62 small
QVTO	Xtend	x ≤ y	0.121	h_0 not rejected	0.63 small
Xtend	ATL	x ≤ y	0.422	h_0 not rejected	0.52 neg.
Change FM					
QVTO	ATL	x ≤ y	0.017	h_0 not rejected	0.71 medium
QVTO	Xtend	x ≤ y	0.032	h_0 not rejected	0.69 medium
Xtend	ATL	x ≤ y	0.376	h_0 not rejected	0.46 neg.
Change RE					
QVTO	ATL	x ≤ y	0.283	h_0 not rejected	0.62 small
QVTO	Xtend	x ≤ y	0.361	h_0 not rejected	0.39 small
Xtend	ATL	x ≤ y	0.029	h_0 not rejected	0.28 medium
Create FM					
QVTO	ATL	x ≤ y	0.241	h_0 not rejected	0.58 small
QVTO	Xtend	x ≤ y	0.203	h_0 not rejected	0.59 small
Xtend	ATL	x ≤ y	0.441	h_0 not rejected	0.51 neg.
Create RE					
QVTO	ATL	x ≤ y	0.038	h_0 not rejected	0.71 medium
QVTO	Xtend	x ≤ y	0.060	h_0 not rejected	0.69 medium
Xtend	ATL	x ≤ y	0.177	h_0 not rejected	0.61 small

The tasks were ordered in increasing cognitive difficulty for all subjects and treatments (comprehend, change, create). Thus, results of the create task are affected by learning effects and fatigue. This is acceptable, as all data points are affected similarly, and because we did not compare performance between tasks. This order also reflects the working situation of developers in practice.

For the comprehension tasks, we traced our observations from the target instance back to the participant's understanding of language concepts. For instance, if no new objects are created, we attributed this to a misunderstanding of the language's concepts for creating objects (e.g., in QVT-O via mapping rules or the `object` keyword). It could be that the participant understood the concept, but assumed it is not used due to misunderstanding control flow. We mitigated this threat by making sure that multiple observations in the target model map to a language concept.

ATL and QVT-O use OCL as an expression language that is known to be complex. We did treat this complexity as an inherent part of both languages. Yet, understanding the impact of this language design choice and comparing OCL to another expression language, would be a valuable future experiment.

External Validity. Naturally, experienced developers working in industry differ from students, even if both have to learn a language completely anew. More experienced developers might reach proficiency faster than students. Yet, they would likely have less time available to learn a language. So, we think that the mistakes and

successes of our students are representative for challenges to be expected from developers who learn a transformation language.

Students might differ due to their demography. All students had at least 3.5 years of university experience including object-oriented programming in Java and programming projects. While the body is relatively uniform, deviations are possible. Our results do not indicate corresponding biases. However, future studies will have to show whether the distribution in experience impacts the results.

The challenges posed in the two devised scenarios may influence the results. To mitigate this, we selected two scenarios representative of common transformations. Still there is substantial risk that some of the observations characterize the scenarios just as much as the languages (especially observations 7–9). Overcoming this risk requires creating even larger studies with more scenarios.

Finally, our observations are stated solely for ATL, QVT-O, and Xtend. We hesitate to generalize them to other languages. Cognition of large non-trivial languages and complex problems is subtle. More studies are needed to establish whether our results pertain to other languages, including GPLs (functional programming languages) and other transformation languages, including grammar [4, 11] and graph-grammar based languages [3, 5, 40]. Our insights about individual language concepts are likely transferable across languages.

Conclusion Validity. A larger sample would strengthen the power of tests and the ability to draw statistically significant conclusions. Yet, it is unlikely to considerably change the effect size. The observed effects are mostly small, with no constancy of medium and large sizes across the tasks and scenarios. For example, there is a large effect for the change task between QVT-O and Xtend in the FM scenario, but only a small one for the RE scenario.

7 RELATED WORK

Hutchinson et al. [15] present results of a twelve-month long empirical study on the application of MDSD in industry. They find that using transformations might increase development time significantly, while decreasing maintenance time due to the possibility of applying transformations multiple times. This acknowledges the relevance of studying transformations languages.

Prior surveys [12, 27] propose taxonomies of model transformations. We selected the subject languages with these taxonomies in mind, sampling over the key dimensions *language paradigm* (declarative vs. imperative) and *application domain* (DSL vs. GPL).

Van Amstel and Van Den Brand [41] introduce and evaluate three analysis techniques for transformation specifications. They suggest that a common understanding of the quality of transformations is missing, which challenges the design of M2M languages. We address this challenge and contribute details about difficult language concepts and challenging transformation skills.

Kosar et al. [21] compare program comprehension between (non-transformation) GPLs and DSLs in three controlled experiments. They find that subjects are more accurate and more efficient in program comprehension when using DSLs. Interestingly, we do not observe the general advantage of transformation DSLs in our experiment, not even just for the comprehension task.

Kramer et al. [22] describe the design of a planned experiment on the comprehension (only) of M2M languages versus GPLs. Unfortunately, no actual experiment is conducted. The scope of our experiment is broader: we also investigate how developers change and create transformations and also compare dedicated M2M languages with each other (not just with a GPL).

Grønmo and Oldevik [13] propose desired properties of M2M languages and use them to compare the UML Model Transformation Tool (UMT) with other languages (including ATL). They solely transform UML, whereas we inspect different model transformation languages irrespective of the transformed language.

Rose et al. [34] report on the model transformation contest[5] where participants perform transformation tasks with various tools. They compare nine different transformation but focus on model migration, so their observations do not easily generalize.

Paige et al. [32] describe state-of-the-art of model evolution. They acknowledge the suitability of M2M languages for this task and name it as a future research direction. Our experiment may help in guiding the search for an ultimate M2M language.

Iosif-Lazar et al. [16] report on implementing and debugging a non-trivial software modernization transformation. They discuss different errors than us, since we observe early stage errors, and they observe errors surviving after a thorough testing process.

8 CONCLUSION

How effectively do programmers use model transformation languages, what problems do they face, and what aspects of these languages emerge as particularly useful or problematic in contrast to GPLs?

All studied languages, declarative, imperative, domain-specific, or general purpose, appear to help users in complex tasks, such as the copying of model parts. The domain-specific languages support subjects better in identifying the starting point and context for changes compared to Xtend (GPL). Still, we did identify a potential for improvement for all studied languages. Most subjects struggled to correctly create and change value conditions, to initialize multi-valued properties, and to use recursion. The future language and tool design research can address some of these problems.

Importantly, we were unable to statistically confirm an anticipated advantage of transformation DSLs over Xtend, or of ATL over the imperative languages. This is a concerning finding: It means that migrating from a GPL to a dedicated transformation language might not bring substantial benefits, especially if size and number of transformations is small (the benefits seen in the experiment are not significant, with small effect sizes). At the same time, expert users of GPLs are much easier to hire than transformation language experts, so productivity with a modern GPL may well be higher. Model transformation researchers, should consider these results, as an indication that further improvements are needed in this technology to warrant strong benefits.

ACKNOWLEDGMENT

We thank our experiment participants, EU H2020 (731869), and the Swedish Research Council (257822902).

REFERENCES

[1] Hervé Abdi. 2007. Bonferroni and Šidák Corrections for Multiple Comparisons. *Encyclopedia of Measurement and Statistics* 3 (2007), 103–107.
[2] Kerstin Altmanninger, Martina Seidl, and Manuel Wimmer. 2009. A Survey on Model Versioning Approaches. *Web Information Systems* 5, 3 (2009), 271–304.

[5] http://www.transformation-tool-contest.eu

[3] Carsten Amelunxen, Alexander Königs, Tobias Rötschke, and Andy Schürr. 2006. MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In *ECMDA-FA'06*.

[4] Jacob Andersen, Claus Brabrand, and David Raymond Christiansen. 2013. Banana Algebra: Compositional Syntactic Language Extension. *Science of Computer Programming* 78, 10 (2013), 1845–1870.

[5] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. 2010. Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In *MODELS'10*.

[6] Thorsten Berger, Divya Nair, Ralf Rublack, Joanne M. Atlee, Krzysztof Czarnecki, and Andrzej Wąsowski. 2014. Three Cases of Feature-Based Variability Modeling in Industry. In *MODELS'14*.

[7] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wąsowski. 2013. A Survey of Variability Modeling in Industrial Practice. In *VaMoS'13*.

[8] Lorenzo Bettini. 2013. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt.

[9] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. 2012. *Model-Driven Software Engineering in Practice*. Morgan & Claypool.

[10] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. 2006. Stratego/XT 0.16: Components for Transformation Systems. In *PEPM'06*.

[11] James R. Cordy. 2006. The TXL Source Transformation Language. *Science of Computer Programming* 61, 3 (2006), 190–210.

[12] Krzysztof Czarnecki and Simon Helsen. 2006. Feature-based Survey of Model Transformation Approaches. *IBM Systems* 45, 3 (July 2006), 621–645.

[13] Roy Grønmo and Jon Oldevik. 2005. An Empirical Study of the UML Model Transformation Tool (UMT). In *INTEROP-ESA'05*. Geneva, Switzerland.

[14] Regina Hebig, Christoph Seidl, Thorsten Berger, John Kook Pedersen, and Andrzej Wąsowski. 2018. M2M Transformation Experiment Online Appendix. (2018). https://sites.google.com/view/m2mappendix

[15] John Hutchinson, Jon Whittle, Mark Rouncefield, and Steinar Kristoffersen. 2011. Empirical Assessment of MDE in Industry. In *ICSE'11*.

[16] Alexandru F. Iosif-Lazar, Ahmad Salim Al-Sibahi, Aleksandar S. Dimovski, Juha Erik Savolainen, Krzysztof Sierszecki, and Andrzej Wasowski. 2015. Experiences from Designing and Validating a Software Modernization Transformation (E). In *ASE'15*.

[17] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, Ivan Kurtev, and Patrick Valduriez. 2006. ATL: a QVT-like Transformation Language. In *OOPSLA'06*.

[18] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report. DTIC Document.

[19] Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. 2009. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *SCAM'09*.

[20] Dimitrios S. Kolovos, Richard F. Paige, and Fiona Polack. 2008. The Epsilon Transformation Language. In *ICMT'08*.

[21] Tomaž Kosar, Marjan Mernik, and Jeffrey C Carver. 2012. Program Comprehension of Domain-specific and General-purpose Languages: Comparison using a Family of Experiments. *Empirical Software Engineering* 17, 3 (2012), 276–304.

[22] Max E. Kramer, Georg Hinkel, Heiko Klare, Michael Langhammer, and Erik Burger. 2016. A Controlled Experiment Template for Evaluating the Understandability of Model Transformation Languages. In *HuFaMo*.

[23] Ivan Kurtev. 2007. State of the Art of QVT: A Model Transformation Language Standard. In *AGTIVE'07*.

[24] Tanja Mayerhofer. 2012. Testing and Debugging UML Models Based on fUML. In *ICSE'12*. IEEE, 1579–1582.

[25] Stephen J Mellor. 2004. *MDA Distilled: Principles of Model-driven Architecture*. Addison-Wesley Professional.

[26] Tom Mens. 2013. *Model Transformation: A Survey of the State of the Art*. John Wiley & Sons, Inc., 1–19.

[27] Tom Mens and Pieter Van Gorp. 2006. A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science* 152 (2006), 125 – 142. GraMoT'05.

[28] Sadaf Mustafiz and Hans Vangheluwe. 2013. Explicit Modelling of Statechart Simulation Environments. In *SummerSim'13*.

[29] Vu Nguyen, Sophia Deeds-Rubin, Thomas Tan, and Barry Boehm. 2007. A SLOC Counting Standard. In *COCOMO II Forum*.

[30] Ulrich Nickel, Jörg Niere, and Albert Zündorf. 2000. The FUJABA Environment. In *ICSE'00*.

[31] Object Management Group. 2014. MDA Guide revision 2.0. http://www.omg.org/cgi-bin/doc?ormsc/14-06-01. (2014).

[32] Richard F. Paige, Nicholas Matragkas, and Louis M. Rose. 2016. Evolving Models in Model-driven Engineering: State-of-the-art and Future Challenges. *Journal of Systems and Software* 111 (2016), 272–280.

[33] John Kook Pedersen. 2013. *Usability and Performance in Model Transformation Languages*. Master's thesis. IT University of Copenhagen.

[34] Louis M. Rose, Markus Herrmannsdoerfer, Steffen Mazanek, Pieter Van Gorp, Sebastian Buchwald, Tassilo Horn, Elina Kalnina, Andreas Koch, Kevin Lano, Bernhard Schätz, et al. 2014. Graph and Model Transformation Tools for Model Migration. *Software & Systems Modeling* 13, 1 (2014), 323–359.

[35] Bran Selic. 2003. The Pragmatics of Model-Driven Development. *IEEE Software* 20, 5 (2003), 19–25.

[36] Anthony M. Sloane and Matthew Roberts. 2015. Oberon-0 in Kiama. *Science of Computer Programming* 114 (2015), 20–32.

[37] Ian Sommerville. 2013. *Software Engineering: Pearson New International Edition*. Pearson Education Limited.

[38] Thomas Stahl and Markus Völter. 2005. *Model-Driven Software Development*. Wiley.

[39] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. 2008. *Eclipse Modeling Framework, 2nd Edition*. Addison Wesley.

[40] Gabriele Taentzer. 2003. AGG: A Graph Transformation Environment for Modeling and Validation of Software. In *AGTIVE'03*.

[41] Marcel van Amstel and Mark van den Brand. 2011. Model Transformation Analysis: Staying ahead of the Maintenance Nightmare. *Theory and Practice of Model Transformations* (2011), 108–122.

[42] András Vargha and Harold D Delaney. 2000. A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong. *Educational and Behavioral Statistics* 25, 2 (2000), 101–132.

[43] Jing Zhang, Yuehua Lin, and Jeff Gray. 2005. *Generic and Domain-specific Model Refactoring using a Model Transformation Engine*. Springer, 199–217.